Combinatorics in PanAxiom

Alasdair McAndrew

November 2015

PanAxiom currenly contains a little combinatorial functionality. The source file `combfunc.spad` contains a few functions for dealing with factorials, and permuations themselves and permutation groups are provided by `perm.spad` abd `permgrp.spad`.

However, there seems to be no functionality for listing permutations. This is the beginning of attempting to address that lack.

We have written a few functions (in the Interactive Language) as a proof of concept. They can be rewritten into a spad file later.

The functions include methods for listing subsets, permutations and derangements, and set partitions;

Subsets

`powerSet` Given a set, produces its power set: the set of all subsets. For example:

```
() -> S := set[4,14,46,5]

   {4,5,14,46}
                                        Type:  Set(PositiveInteger)

() -> powerSet(S)

{{}, {4}, {5}, {5,4}, {14}, {14,4}, {14,5}, {14,5,4}, {46},
{46,4}, {46,5}, {46,5,4}, {46,14}, {46,14,4}, {46,14,5},
{46,14,5,4}}
                                                Type:  Set(Set(Any))
```

There is no restriction on the elements of the set:

```
() -> S:=set["A",2.5,vector[1,2,3]]

   {"A",2.5,[1,2,3]}
                                                     Type:  Set(Any)

() -> powerSet(S)

{{}, {"A"}, {2.5}, {2.5,"A"}, {[1,2,3]}, {[1,2,3],"A"},
{[1,2,3],2.5}, {[1,2,3],2.5,"A"}}
                                                Type:  Set(Set(Any))
```

`choose` The function `choose(S,n)` lists all subsets of S containing n elements. There is no restriction on elements of S:

```
() -> S:=set["cat","dog","fly","eel"]
    {"cat","dog","eel","fly"}
                                                Type:  Set(String)
() -> choose(S,2)
{{"eel","fly"}, {"dog","fly"}, {"dog","eel"}, {"cat","fly"},
{"cat","eel"}, {"cat","dog"}}
                                            Type:  Set(Set(Any))
```

Permutations

`listPermutations` Given a list containing any elements, including repeated elements, lists all permutations. The algorithm used is called by Knuth (TAOCP) "Algorithm L", and lists all permutations in lexicographical order. The first permutation consists of the list with elements in non-decreasing order, and the steps to move from one permutation p with n elements to the next consist of

1. find the largest k such that $p.k > p.(k+1)$
2. find the largest j such that $p.j > p.k$
3. swap $p.j$ and $p.k$
4. reverse that part of the list from indices $k+1$ to n.

For example:

```
() -> L:=[1,1,2,3,3];
() -> listPermutations(L)
[[1,1,2,3,3], [1,1,3,2,3], [1,1,3,3,2], [1,2,1,3,3], [1,2,3,1,3],
 [1,2,3,3,1], [1,3,1,2,3], [1,3,1,3,2], [1,3,2,1,3], [1,3,2,3,1],
 [1,3,3,1,2], [1,3,3,2,1], [2,1,1,3,3], [2,1,3,1,3], [2,1,3,3,1],
 [2,3,1,1,3], [2,3,1,3,1], [2,3,3,1,1], [3,1,1,2,3], [3,1,1,3,2],
 [3,1,2,1,3], [3,1,2,3,1], [3,1,3,1,2], [3,1,3,2,1], [3,2,1,1,3],
 [3,2,1,3,1], [3,2,3,1,1], [3,3,1,1,2], [3,3,1,2,1], [3,3,2,1,1]]
                                                List(List(Any))
```

`randomPermutation` This uses Knuth's shuffle algorithm; for a list L with n elements, for each i from 1 to $n-1$ it chooses a random element from the list, and swaps it with $L[i]$.

Derangements

A *derangement* is a permutation where none of the elements are in their original place. For example, here is a listing of the derangements of $1, 2, 3, 4$:

1	2	3	4
2	1	4	3
2	3	4	1
2	4	1	3
3	1	4	2
3	4	1	2
3	4	2	1
4	1	2	3
4	3	1	2
4	3	2	1

It can be shown that if D_n is the number of derangements, then

$$D_n = n\left(\frac{1}{0!} - \frac{1}{1!} + \frac{1}{2!} - \cdots + (-1)^n \frac{1}{n!}\right)$$
$$D_n = nD_{n-1} + (-1)^n$$
$$D_n = (n-1)(D_{n-1} + D_{n-2})$$

We see from the above listing that $D_4 = 9$.

`derangements(n)` produces the number of derangements of n distinct objects. We use the last property above, along with the starting values $D_1 = 0$ and $D_2 = 1$.

`listDerangements(L)` lists all the derangements of a list, by iterating through the list of all permutations, and only keeping those which have no fixed points. Since

$$\lim_{n\to\infty} \frac{D_n}{n!} = e^{-1}$$

this is reasonably efficient.

For example, with the same list as above:

```
() -> listDerangements(L)
[[2,3,3,1,1],[3,2,3,1,1],[3,3,1,1,2],[3,3,1,2,1]]
                                        List(List(Any))
```

`countDerangements(L)` This produces the number of derangements of any list. If there are k distinct elements of L have multiplicities $a_1, a_2, \ldots, a_k$, then the number of derangements was first published by Major Percy MacMahon in "Combinbatory Analysis" (1915):

Let $x_1, x_2, \ldots, x_k$ be variables and put $S = x_1 + x_2 + \cdots + x_k$. Then the required number of derangements is the coefficient of

$$x_1^{a_1} x_2^{a_2} \ldots x_k^{a_k}$$

in the expansion of

$$(S - x_1)^{a_1} (S - x_2)^{a_2} \ldots (S - x_k)^{a_k}.$$

Note that if any one a_j satisfies $2a_j > n$, where n is the length of the list L, then there will be zero derangements.

`randomDerangement(L)` This is a probabilistic function: it chooses permutations at random until on of them is a derangement. If the number of derangements is known to be zero, the function halts with an error message.

The permutations and derangement functions are also implemented for strings:

```
() -> listStringPermutations("EERIE")

["EEERI", "EEEIR", "EEREI", "EERIE", "EEIER", "EEIRE", "EREEI",
 "EREIE", "ERIEE", "EIEER", "EIERE", "EIREE", "REEEI", "REEIE",
 "REIEE", "RIEEE", "IEEER", "IEERE", "IEREE", "IREEE"]
                                                    List(String)
() -> listStringDerangements("banana")

["abanan","anaban","ananab"]
                                                    List(String)
```

Set partitions

A *partition* of a set S is a splitting of S into a set of subsets T_i of S which are pairwise disjoint, and whose union is S. For example, a three element set has five partitions:

$$\{1,2,3\} \to \{\{1,2,3\}, \{\{1\},\{2,3\}\}, \{\{2\},\{1,3\}\}, \{\{3\},\{1,2\}\}, \{\{1\},\{2\},\{3\}\}\}.$$

There is one partition into one subset, three partitions into two subsets, and one partitions into three subsets. The number of partitions of an n element set S into k subsets is given by the Stirling number of the second kind $S_2(n,k)$. This is already implemented:

```
() -> stirling2(3,2)
   3
                                        Type: PositiveInteger
```

The sum of all Stirling numbers of the second kind for a given n is called the *Bell number*, denoted B_n. Using Knuth's bracketing notation $\left[{n \atop k}\right]$ for $S_2(n,k)$, we have

$$B_n = \sum_{k=0}^{n} \left[{n \atop k}\right].$$

Stirling numbers of the second kind can also be given by using *Touchard polynomials* as generating functions. These can be defined recursively by

$$T_0(x) = 1$$

$$T_n(x) = x\left(1 + \frac{d}{dx}\right) T_{n-1}(x) \text{ for } n \geq 1$$

Then $\left[{n \atop k}\right]$ is the coefficient of x^k in $T_n(x)$.

Here are the functions:

`listPartitions` This function lists all partitions as codes, so that for example, if $S = \{2, 4, 6, 8, 10\}$ and the code is $[1, 2, 1, 3, 1]$ then the corresponding subset partition is $\{\{2, 6, 10\}, \{4\}, \{8\}\}$. That is, element S_i of S belongs to subset C_i.

```
() -> listPartitions(3)
    [[1,1,1],[1,1,2],[1,2,1],[1,2,2],[1,2,3]]
                                        Type:  List(List(Integer))
```

`setPartitions` returns the partitions of a set, by mapping the elements of the set onto the codes generated by the previous algorithm.

```
() -> S:Set Any:=set[2.5,'x+'y,"A"]
() -> setPartitions(S)
{{{2.5,y + x,"A"}}, {{2.5,y + x},{"A"}}, {{2.5,"A"},{y + x}},
{{2.5},{y + x,"A"}}, {{2.5},{y + x},{"A"}}}
                                        Type:  Set(Set(Any))
```

`listSizePartitions(n,k)` is a slight variant of `listPartitions` and lists only those codes whose maximum value is k. These will correspond to partitions into exactly k subsets.

```
() -> P:=listSizePartitions(5,3)
[[1,1,1,2,3], [1,1,2,1,3], [1,1,2,2,3], [1,1,2,3,1], [1,1,2,3,2],
 [1,1,2,3,3], [1,2,1,1,3], [1,2,1,2,3], [1,2,1,3,1], [1,2,1,3,2],
 [1,2,1,3,3], [1,2,2,1,3], [1,2,2,2,3], [1,2,2,3,1], [1,2,2,3,2],
 [1,2,2,3,3], [1,2,3,1,1], [1,2,3,1,2], [1,2,3,1,3], [1,2,3,2,1],
 [1,2,3,2,2], [1,2,3,2,3], [1,2,3,3,1], [1,2,3,3,2], [1,2,3,3,3]]
                                        List(List(Integer))
```

The number of such partitions should be the Stirling number $\left[{n \atop k}\right]$:

```
() -> #P
    25
                                        Type:  PositiveInteger
() -> stirling2(5,3)
    25
                                        Type:  PositiveInteger
```

`setSizePartitions(S,k)` turns the list of codes into partitions of S into k subsets.

`bell(n)` returns the n-th Bell number

`touchard(n,x)` returns the n-th Touchard polynomial:

```
() -> p := touchard(7,x)

 7      6       5       4       3      2
x  + 21x  + 140x  + 350x  + 301x  + 63x  + x

                                Type:  Polynomial(Fraction(Integer))
```

We can check that the coefficients are the Stirling numbers:

```
() -> [coefficient(p,x,k) for k in 0..7]
     [0,1,63,301,350,140,21,1]
                        Type:  List(Polynomial(Fraction(Integer)))
() -> [stirling2(7,k) for k in 0..7]
     [0,1,63,301,350,140,21,1]
                                                Type:  List(Integer)
```

The values are the same, although the lists are of different types.